



中山大学 软件工程学院
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

《Rust语言与内存安全设计》

第9讲 Rust与函数式编程

课程负责人：陈文清 助理教授
`chenwq95@mail.sysu.edu.cn`

2023年10月25日

目录













1 习题讲解

2 Rust中的函数式编程语言特性

泛型、特征、生命周期：课堂练习



文件夹	其他
 常见集合	 generics1.rs
 错误控制	 generics2.rs
 泛型_特征_生命周期	 lifetimes1.rs
 所有权	 lifetimes2.rs
	 traits1.rs
	 traits2.rs

泛型、特征、生命周期：课堂练习



目录



1 习题讲解

2 Rust中的函数式编程语言特性

函数式编程


在这个Java示例中，我们使用**Lambda表达式**和Stream API，通过.map()方法将每个元素平方

然后使用.forEach()方法遍历并打印结果。

接着，我们使用.reduce()方法计算整数列表的总和。

这个示例展示了Java中的函数式编程特性，以一种更函数式的方式处理数据集合。

java

 Copy code

```
import java.util.Arrays;
import java.util.List;

public class FunctionalProgrammingExample {
    public static void main(String[] args) {
        // 创建一个整数列表
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // 使用Stream API和Lambda表达式将每个元素平方并打印出来
        numbers.stream()
            .map(number -> number * number)
            .forEach(System.out::println);

        // 使用Stream API和Lambda表达式计算列表中所有元素的总和
        int sum = numbers.stream()
            .reduce(0, (x, y) -> x + y);
        System.out.println("Sum: " + sum);
    }
}
```

函数式编程


在这个Python示例中，我们使用**Lambda表达式**来定义一个简单的函数，

并使用map函数将该函数应用于列表中的每个元素，

然后使用reduce函数计算列表中所有元素的总和。

函数式编程的一个关键思想是避免可变状态，以便更容易理解和测试代码。

python

 Copy code

```
# 使用lambda函数定义一个匿名函数，该函数计算一个数的平方
```

```
square = lambda x: x**2
```

```
# 创建一个整数列表
```

```
numbers = [1, 2, 3, 4, 5]
```

```
# 使用map函数将square函数应用于列表中的每个元素
```

```
squared_numbers = list(map(square, numbers))
```

```
# 打印平方后的结果
```

```
print(squared_numbers) # 输出 [1, 4, 9, 16, 25]
```

```
# 使用reduce函数计算列表中所有元素的和
```

```
from functools import reduce
```

```
total = reduce(lambda x, y: x + y, numbers)
```

```
# 打印总和
```

```
print(total) # 输出 15
```

Rust的设计受到了许多现有语言和技术启发，其中一个重要的影响是**函数式编程**。以函数式风格编程通常包括**将函数用作值**，通过**将它们作为参数传递**，从其他函数返回它们，**将它们分配给变量**以便以后执行等操作。

具体来说，我们将涵盖以下内容：

- 1. 闭包 (Closures)**，一种类似函数的构造，可以存储在变量中
- 2. 迭代器 (Iterator)**，一种处理一系列元素的方法

我们已经介绍了一些其他Rust特性，例如模式匹配和枚举，它们也受到了函数式风格的影响。因为掌握闭包和迭代器是编写符合惯例且高效的Rust代码的重要组成部分。

什么是闭包?

闭包



■ 什么是闭包：

- Rust的闭包是匿名函数
- 可以将它们保存在变量中或将它们作为参数传递给其他函数。
- 可以在一个地方创建闭包，然后在不同的上下文中调用闭包以对其进行评估。
- 与函数不同，闭包可以捕获其定义的作用域中的值。

闭包



■ 什么是闭包：

- Rust的闭包是匿名函数
- 可以将它们保存在变量中或将它们作为参数传递给其他函数。
- 可以在一个地方创建闭包，然后在不同的上下文中调用闭包以对其进行评估。
- 与函数不同，闭包可以捕获其定义的作用域中的值。

```
fn  add_one_v1    (x: u32) -> u32 { x + 1 }  
let add_one_v2 = |x: u32| -> u32 { x + 1 };  
let add_one_v3 = |x|           { x + 1 };  
let add_one_v4 = |x|           x + 1 ;
```

闭包



■ 什么是闭包：

- Rust的闭包是匿名函数
- 可以将它们保存在变量中或将它们作为参数传递给其他函数。
- 可以在一个地方创建闭包，然后在不同的上下文中调用闭包。
- 与函数不同，闭包可以捕获其定义的作用域中的值。

闭包



■ 什么是闭包:

```
fn main(){  
  
    fn add_one_v1    (x: u32) -> u32 { x + 1 }  
    let add_one_v2: impl Fn(u32) -> u32 = |x: u32| -> u32 { x + 1 };  
    let add_one_v3: impl Fn(i32) -> i32 = |x: i32| { x + 1 };  
    let add_one_v4: impl Fn(i32) -> i32 = |x: i32| x + 1 ;  
  
    let res1: u32 = add_one_v1(3);  
    let res2: u32 = add_one_v2(3);  
    let res3: i32 = add_one_v3(3);  
    let res4: i32 = add_one_v4(3);  
  
    println!("{}", {}, {}, {}, {}, res1, res2, res3, res4);  
}
```

闭包



■ 什么是闭包：

- Rust的闭包是匿名函数
- 可以将它们保存在变量中或将它们作为参数传递给其他函数。
- 可以在一个地方创建闭包，然后在不同的上下文中调用闭包。
- 与函数不同，闭包可以捕获其定义的作用域中的值。

闭包



■ 什么是闭包:

➤ Rust的

➤ 可以

➤ 可以

➤ 与函数

```
fn main(){  
  
    let add_num: i32 = 4;  
    fn add_one_v1 (x: u32) -> u32 { x + add_num }  
    let add_one_v2: impl Fn(u32) -> u32 = |x: u32| -> u32 { x + 1 };  
    let add_one_v3: impl Fn(i32) -> i32 = |x: i32| { x + 1 };  
    let add_one_v4: impl Fn(i32) -> i32 = |x: i32| x + add_num ;  
  
    let res1: u32 = add_one_v1(3);  
    let res2: u32 = add_one_v2(3);  
    let res3: i32 = add_one_v3(3);  
    let res4: i32 = add_one_v4(3);  
  
    println!("{}", {}, {}, {}, {}, res1, res2, res3, res4);  
}
```

普通的函数无法利用其上下文的值，而闭包可以

闭包

■ 什么是闭包:

- Rust的闭包
- 可以将它
- 可以在一
- 与函数不

```
#[derive(Debug, PartialEq, Copy, Clone)]
enum ShirtColor {
    Red,
    Blue,
}

struct Inventory {
    shirts: Vec<ShirtColor>,
}

impl Inventory {
    fn giveaway(&self, user_preference: Option<ShirtColor>) -> ShirtColor {
        user_preference.unwrap_or_else(|| self.most_stocked())
    }

    fn most_stocked(&self) -> ShirtColor {
        let mut num_red = 0;
        let mut num_blue = 0;

        for color in &self.shirts {
            match color {
                ShirtColor::Red => num_red += 1,
                ShirtColor::Blue => num_blue += 1,
            }
        }
        if num_red > num_blue {
            ShirtColor::Red
        } else {
            ShirtColor::Blue
        }
    }
}
```

此处的闭包使用了
闭包外部环境的变量**self**



闭包的类型推断

■ 闭包的类型推断:

- 函数和闭包之间有更多的区别。
- 通常情况下，闭包不需要你像函数一样注释参数或返回值的类型。
- 但函数需要注释参数或返回值的类型，因为类型是显式接口的一部分，向用户公开。严格定义这个接口对于确保每个人都同意函数使用和返回的值类型非常重要。
- 另一方面，闭包不像这样在公开接口中使用：它们存储在变量中，可以在不命名它们或不向用户公开它们的情况下使用。

闭包



■ 闭包的类型推断：

➤ 但是，省略注释有时也会存在问题，请思考以下代码能否编译通过？

```
let example_closure = |x| x;  
  
let s = example_closure(String::from("hello"));  
let n = example_closure(5);
```

闭包中的引用与所有权

闭包



■ 捕获引用或转移所有权：

- 闭包可以以三种方式捕获其环境中的值，这直接对应于函数可以接受参数的三种方式：
不可变地借用、可变地借用和接管所有权。

闭包



■ 捕获引用或转移所有权：

- 闭包可以以三种方式捕获其环境中的值，这直接对应于函数可以接受参数的三种方式：
不可变地借用、可变地借用和接管所有权。

```
fn main() {  
    let list = vec![1, 2, 3];  
    println!("Before defining closure: {:?}", list);  
  
    let only_borrows = || println!("From closure: {:?}", list);  
  
    println!("Before calling closure: {:?}", list);  
    only_borrows();  
    println!("After calling closure: {:?}", list);  
}
```

■ 捕获引用或转移所有权：

- 闭包可以以三种方式捕获其环境中的值，这直接对应于函数可以接受参数的三种方式：不可变地借用、可变地借用和接管所有权。

```
fn main() {  
    let mut list = vec![1, 2, 3];  
    println!("Before defining closure: {:?}", list);  
  
    let mut borrows_mutably = || list.push(7);  
  
    borrows_mutably();  
    println!("After calling closure: {:?}", list);  
}
```

闭包



■ 捕获引用或转移所有权：

- 闭包可以以三种方式捕获其环境中的值，这直接对应于函数可以接受参数的三种方式：不可变地借用、可变地借用和接管所有权。

```
fn main() {  
    let list: Vec<i32> = vec![1, 2, 3];  
    println!("Before defining closure: {:?}", list);  
  
    let test_closure: impl FnOnce() -> Vec<i32> = || list;  
  
    let new_list: Vec<i32> = test_closure();  
  
    println!("After running closure: {:?}", list);  
}
```


闭包



■ 捕获引用或转移所有权：

- 闭包可以以三种方式捕获其环境中的值，这直接对应于函数可以接受参数的三种方式：不可变地借用、可变地借用和接管所有权。

```
use std::thread;

fn main() {
    let list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);

    thread::spawn(move || println!("From thread: {:?}", list))
        .join()
        .unwrap();
}
```

将从闭包中捕获的值移出，以及Fn特征

闭包



■ 回顾什么是闭包：

- Rust的闭包是匿名函数
- 可以将它们保存在变量中或将它们作为参数传递给其他函数（也可作为返回值）。
- 可以在一个地方创建闭包，然后在不同的上下文中调用闭包以对其进行评估。
- 与函数不同，闭包可以捕获其定义的作用域中的值。

闭包



■ 将从闭包中捕获的值移出，以及Fn特征：

- Rust的闭包是匿名函数
- 可以将它们保存在变量中或将它们作为参数传递给其他函数。
- 可以在一个地方创建闭包，然后在不同的上下文中调用闭包以对其进行评估。
- 与函数不同，闭包可以捕获其定义的作用域中的值。

■ Fn, FnMut和FnOnce特征 (Trait) :

- 闭包从环境中捕获和处理值的方式会影响它实现的特征 (trait) ;
- 闭包将根据其体中的值的处理方式自动实现这三个Fn特质之一、两个或全部三个:
 - FnOnce 适用于**只能调用一次**的闭包。所有闭包至少实现了这个特征，因为所有闭包都可以被调用。
 - FnMut 适用于**不会从闭包中移出捕获值，但可能会更改捕获值的闭包**。这些闭包可以被多次调用。
 - Fn 适用于**不会从其体中移出捕获值，不会更改捕获值的闭包**，以及不从其环境中捕获任何内容的闭包。这些闭包可以被多次调用而不会更改其环境。

闭包



■ Fn, FnMut和FnOnce特征 (Trait) :

```
impl<T> Option<T> {  
    pub fn unwrap_or_else<F>(self, f: F) -> T  
    where  
        F: FnOnce() -> T  
    {  
        match self {  
            Some(x) => x,  
            None => f(),  
        }  
    }  
}
```

闭包



■ Fn, FnMut和FnOnce特征 (Trait) :

```
fn feel_lucky() -> Option<String> {  
    if get_random_number() > 50 {  
        Some(String::from("feel lucky"))  
    }else{  
        None  
    }  
}
```

```
fn main() {  
  
    let message: String = feel_lucky().unwrap();  
    let message: String = feel_lucky().unwrap_or(default: String::from("feel unlucky"));  
    let message: String = feel_lucky().unwrap_or_else(|| String::from("feel unlucky"));  
    println!("{}", message);  
}
```

闭包



■ Fn, FnMut和FnOnce特征 (Trait) :

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    list.sort_by_key(|r| r.width);
    println!("{:?}", list);
}
```

https://docs.rs/radsort/latest/radsort/fn.sort_by_key.html

闭包



■ Fn, FnMut和FnOnce特征 (Trait) :

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let mut list = [
        Rectangle { width: 10, height: 1 },
        Rectangle { width: 3, height: 5 },
        Rectangle { width: 7, height: 12 },
    ];

    let mut num_sort_operations = 0;
    list.sort_by_key(|r| {
        num_sort_operations += 1;
        r.width
    });
    println!("{:?}", sorted in {num_sort_operations} operations", list);
}
```

迭代器 (Iterator)



什么是迭代器?

迭代器



■ 迭代器 (Iterator)

- 迭代器允许依次对一系列项 (**item**) 执行某些任务
- 迭代器负责:
 - 遍历每个项 (**item**) ;
 - 确定遍历何时完成。
- Rust中的迭代器特点:
 - 懒惰的 (**lazy**) : 这意味着在你调用消耗迭代器的方法之前, 它们不产生任何作用;

迭代器



■ 迭代器 (Iterator)

➤ Rust中的迭代器特点:

➤ 懒惰的 (lazy): 这意味着在你调用消耗迭代器的方法之前, 它们不产生任何作用;

```
fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
  
    let v1_iter: Iter<i32> = v1.iter();  
}
```

迭代器



■ 迭代器 (Iterator)

```
fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
  
    let v1_iter: Iter<i32> = v1.iter();  
  
    for val: &i32 in v1_iter {  
        println!("Got: {}", val);  
    }  
}
```

迭代器



■ 迭代器 (Iterator)

```
fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
  
    let v1_iter: Iter<i32> = v1.iter();  
  
    for val: &i32 in v1_iter {  
        println!("Got: {}", val);  
    }  
}
```

在没有提供迭代器的语言中，你可能会编写相同的功能，方法是：

在索引0处开始一个变量，使用该变量索引向量以获取一个值，并在循环中递增变量值 直到达到向量中的item总数。

因此Iterator的作用之一是减少repetitive code。

迭代器



■ Iterator trait

- 定义于标准库;
- type Item是声明某种类型;
- Iterator trait要求实现next方法

```
pub trait Iterator {  
    type Item;  
  
    fn next(&mut self) -> Option<Self::Item>;  
  
    // methods with default implementations elided  
}
```

迭代器



■ Iterator trait

- 定义于标准库;
- type Item是声明某种类型;
- Iterator trait要求实现next方法

```
#[test]
▶ Run Test | Debug
fn iterator_demonstration() {
    let v1: Vec<i32> = vec![1, 2, 3];

    let mut v1_iter: Iter<i32> = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}
```


迭代器



■ Iterator trait

- 定义于标准库;
- type Item是声明某种类型;
- Iterator trait要求实现next方法

注意**mut**关键词
(调用**next**方法时会消耗调用次数)

```
#[test]
▶ Run Test | Debug
fn iterator_demonstration() {
    let v1: Vec<i32> = vec![1, 2, 3];

    let mut v1_iter: Iter<i32> = v1.iter();

    assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}
```

迭代器



■ Iterator trait

- 定义于标准库;
- type Item是声明某种类型;
- Iterator trait要求实现next方法

注意**mut**关键词

(调用**next**方法时会消耗调用次数)

```
fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
  
    let v1_iter: Iter<i32> = v1.iter();  
  
    for val: &i32 in v1_iter {  
        println!("Got: {}", val);  
    }  
}
```

} **for**循环并不需要添加**mut**关键词

因为**for**循环已经取得**v1_iter**的所有权,

在内部把它变为**mutable**。(for循环后无法再调用**v1_iter**)

#[test]

▶ Run Test | Debug

```
fn iterator_demonstration() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
  
    let mut v1_iter: Iter<i32> = v1.iter();  
  
    assert_eq!(v1_iter.next(), Some(&1));  
    assert_eq!(v1_iter.next(), Some(&2));  
    assert_eq!(v1_iter.next(), Some(&3));  
    assert_eq!(v1_iter.next(), None);  
}
```

迭代器



■ 几种迭代方法（后两种请大家查阅API学习）：

- iter方法：在不可变引用上创建迭代器；
- into_iter方法：创建的迭代器会获得所有权；
- iter_mut方法：迭代可变的引用。

迭代器



■ 几种迭代方法:

➤ into_iter方法: 创建的迭代器会获得所有权;

► Run | Debug

```
fn main() {  
    let v1: Vec<i32> = vec![1, 2, 3];  
  
    let v1_iter: IntoIter<i32> = v1.into_iter();  
  
    for v1: i32 in v1_iter {  
        println!("{}", v1);  
    }  
  
    println!("{:?}", v1);  
}
```

→ into_iter方法

→ v1此时不再是*&i32*类型

→ 报错: value borrowed here after move

迭代器



■ 几种迭代方法:

➤ `iter_mut`方法: 迭代可变的引用;

```
fn main() {  
    let mut v1: Vec<i32> = vec![1, 2, 3];  
  
    let v1_iter: IterMut<i32> = v1.iter_mut();  
  
    for v1: &mut i32 in v1_iter {  
        *v1 += 1;  
        println!("{}", v1);  
    }  
  
    println!("{:?}", v1);  
}
```

➔ 添加**mut**关键词

➔ **iter_mut**方法

➔ 可修改**vector**里的元素

➔ 打印结果: [2, 3, 4]

第二次作业



题目：几何形状管理程序（考察Struct、Trait、Generic的用法）

要求：

1. 创建一个名为Shape的Trait，其中包括以下方法：
 1. `area(&self) -> f64`：计算几何形状的面积。
 2. `perimeter(&self) -> f64`：计算几何形状的周长。
2. 创建三个Struct，分别代表以下几何形状，每个Struct都必须实现Shape Trait：
 1. 矩形（Rectangle）：包含长度和宽度。
 2. 圆形（Circle）：包含半径。
 3. 三角形（Triangle）：包含三条边的长度。
3. 创建一个泛型函数`print_shape_info<T: Shape>(shape: T)`，它接受任何实现了Shape Trait的几何形状，然后打印该几何形状的类型、面积和周长。
4. 在main函数中，创建至少一个矩形、一个圆形和一个三角形的实例，并使用`print_shape_info`函数分别输出它们的信息。

第二次作业



题目：几何形状管理程序（考察Struct、Trait、Generic的用法）

要求：

1. 创建一个名为Shape的Trait，其中包括以下方法：

1. `area(&self) -> f64`：计算几何形状的面积。
2. `perimeter(&self) -> f64`：计算几何形状的周长。

2. 创建三个Struct，分别代表以下几何形状，每个Struct都必须实现Shape Trait：

1. 矩形（Rectangle）：包含长度和宽度。
2. 圆形（Circle）：包含半径。
3. 三角形（Triangle）：包含三条边的长度。

3. 创建一个泛型函数`print_shape_info<T: Shape>(shape: T)`，它接受任何实现了Shape Trait的几何形状，然后打印该几何形状的类型、面积和周长。

4. 在main函数中，创建至少一个矩形、一个圆形和一个三角形的实例，并使用`print_shape_info`函数分别输出它们的信息。

提示：

- 在Shape Trait中，你可以使用关联类型来定义几何形状的属性，例如面积和周长的类型。这样，每个实现Trait的类型可以定义自己的关联类型。
- 在main函数中，可以使用泛型函数`print_shape_info`来减少代码重复。

这个作业将考察学生对Trait、Struct、以及泛型的理解和运用。学生需要创建自定义的Struct，并确保它们实现了Trait中定义的方法。同时，他们需要编写一个泛型函数来处理不同类型的几何形状，并灵活地使用Trait来计算面积和周长。这有助于加强对Rust中泛型和Trait系统的理解。



中山大學
SUN YAT-SEN UNIVERSITY

Q & A

Thanks!